On the numerical reliability of nonsmooth autodiff: a MaxPool case study

Anonymous authors
Paper under double-blind review

Abstract

This paper considers the reliability of automatic differentiation for neural networks involving the nonsmooth MaxPool operation across various precision levels (16, 32, 64 bits), architectures (LeNet, VGG, ResNet), and datasets (MNIST, CIFAR10, SVHN, ImageNet). Although AD can be incorrect, recent research has shown that it coincides with the derivative almost everywhere, even in the presence of nonsmooth operations (such as MaxPool and ReLU). On the other hand, in practice, AD operates with floating-point numbers, and there is, therefore, a need to explore subsets on which AD can be *numerically* incorrect. These subsets include a bifurcation zone (where AD is incorrect over reals) and a compensation zone (where AD is incorrect over floating-point numbers but correct over reals). Using SGD for the training process, we study the impact of different choices of the nonsmooth Jacobian for the MaxPool function on the precision of 16 and 32 bits. These findings suggest that nonsmooth MaxPool Jacobians with lower norms help maintain stable and efficient test accuracy, whereas those with higher norms can result in instability and decreased performance. We also observe that the influence of MaxPool's nonsmooth Jacobians on learning can be reduced by using batch normalization, Adam-like optimizers, or increasing the precision level.

1 Introduction

Nonsmooth neural networks are trained using optimization algorithms (Bottou et al., 2018; Davis et al., 2018) based on backpropagation and automatic differentiation (AD) (Speelpenning, 1980; Rumelhart et al., 1986b; Baydin et al., 2018). AD is a crucial tool in contemporary learning architectures as it allows for fast differentiation (Griewank & Walther, 2008; Bolte et al., 2022). It is implemented in popular machine learning libraries such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and Jax (Bradbury et al., 2018). Although the validity domain of AD is theoretically limited to smooth functions (Griewank & Walther, 2008), it is commonly used for nonsmooth functions (Bolte et al., 2022; 2021b; Bertoin et al., 2023). The behavior of nonsmooth AD has been investigated in previous studies (Griewank & Walther, 2008; Griewank, 2013; Griewank et al., 2016; Barton et al., 2018; Kakade & Lee, 2018; Griewank & Rojas, 2019; Griewank & Walther, 2020; Bolte & Pauwels, 2020a; Bolte et al., 2022).

MaxPool: a nonsmooth operation Introduced by Yamaguchi et al. (1990), MaxPool is a common operation in convolutional neural networks (CNN), which are a type of network often used for image classification (Krizhevsky & Hinton, 2010; Krizhevsky et al., 2012; Zeiler & Fergus, 2014; LeCun et al., 2015). MaxPool reduces the spatial dimensions of a feature map by selecting

the maximum value within specific patches. MaxPool applied to uniform pixel values can cause nonsmoothness, especially at image edges where identical pixels can be chosen arbitrarily. In such cases, different choices of MaxPool's nonsmooth Jacobians have a variational sense. See Appendix A.2 for an illustration. In this paper, the term *MaxPool-derived program* refers to a specific choice of a MaxPool nonsmooth Jacobian.

Various types of nonsmooth AD errors: We carry out a small PyTorch (Paszke et al., 2019) experiment to investigate the autodiff behavior of the nonsmooth max function, defined as $\max\colon x \mapsto \max_{1\leq i\leq 4} x_i \in \mathbb{R}$. We implement two max programs ($\max_1$ and $\max_2$) with different derivative implementations (see Appendix A.1 for more details). Let zero be a program as follows: $\text{zero}\colon t \mapsto \max_1(t \times x) - \max_2(t \times x)$. The associated AD output of zero is denoted by zero'. As mathematical functions, both $\max_1$ and $\max_2$ output the same value, while zero always outputs 0. However, when using AD and floating-point numbers, we observe an unexpected behavior: $\text{zero}'(t) \neq 0$ for some $t \in \mathbb{R}$. In Table 1, we analyze numerical AD errors for the zero program. For

		$\text{zero}'(t)$						
t		-10^{-3}	-10^{-2}	-10^{-1}	0	10^{1}	10^{2}	10^{3}
$x_1 =$	1.0 2.0 3.0 4.0	0.0	0.0	0.0	-1.5	0.0	0.0	0.0
$x_2 =$	1.4 1.4 1.4 1.4	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}

Table 1: Overview of numerical AD errors for the zero program with 32 bits precision.

x_1, we find a large error for $t = 0$ ($\text{zero}'(0) = -1.5$), which is different from the correct derivative. For x_2, a case with equal numbers common in tasks like image classification (see Appendix A.2), theoretical calculations give $\text{zero}'(t) = 0$ for any $t \in \mathbb{R}$. However, using floating-point numbers, we see small divergences in AD results. Specifically, for all t in Table 1, $\text{zero}'(t)$ is about 5.96×10^{-8} (shown as 10^{-7} in the table), near the limit for 32-bit precision. This phenomenon occurs due to numerical arithmetic limits. In general, t represents a neural network parameter, and x is an input image with a specific pixel area with identical values (e.g., MNIST dataset- refer to Appendix A.2). Note that these phenomena observed in Table 1 are not caused by the nonsmooth multivariate nature of the max function and can also be replicated using only the nonsmooth univariate ReLU operation. Refer to Appendix A.3 for more details.

Reals vs floating-point numbers: Over reals, AD outputs derivatives for nondifferentiable functions, except for a Lebesgue measure-zero subset of inputs (Bolte & Pauwels, 2020a;b). On the other hand, as reported in Table 1, floating-point arithmetic can thicken subsets where AD is incorrect (Bertoin et al., 2023). In Section 3, we try to identify two network parameter subsets where AD is incorrect numerically: the *bifurcation zone* with considerable amplitude variations of AD and the *compensation zone* with minor amplitude variations near machine precision, which is due to rounding schemes used for inexact arithmetics over the reals (e.g., non-associativity). Our experiments show that in a 64-bit network with MaxPool, the compensation zone fills the entire parameter space. In a 32-bit network, both compensation and bifurcation zones share the space. In a 16-bit setting, the bifurcation zone takes up the entire parameter space.

Implications for learning dynamics: In Section 4, we explore how different nonsmooth MaxPool Jacobians affect learning. Using 32-bit precision, nonsmooth Jacobians with low norm yield similar test accuracy. However, Jacobians with high norms decrease accuracy due to training instability or gradient problems. In 16-bit precision, which is a key area of study (Vanhoucke et al., 2011; Hwang

& Sung, 2014; Courbariaux et al., 2015; Gupta et al., 2015), the impact of these Jacobians is more pronounced and varies with the network design, data, and precision used. We also observe that both batch normalization (Ioffe & Szegedy, 2015) and the Adam optimizer (Kingma & Ba, 2014) mitigate this effect. All experiments were done using PyTorch (Paszke et al., 2019), and our code is publicly available [1].

Related works and contributions: Recent works show that for a broad class of programs using nonsmooth functions, AD is incorrect at most on a Lebesgue measure-zero subset of the input domain of a program (Bolte & Pauwels, 2020a; Lee et al., 2020). However, practical inputs are machine-representable. Lee et al. (2023) recently explored AD correctness in neural networks with machine-representable parameters, excluding networks with MaxPool. Bertoin et al. (2023) examined the $\mathrm{ReLU}'(0)$ effect on AD and training, identifying a *bifurcation zone* in ReLU networks where AD fails. However, they do not consider the case, where AD is incorrect over floating-point numbers but correct over real numbers (e.g., last line in Table 1). Thus, our paper introduces a *compensation zone*. Our study assesses autodiff reliability in MaxPool neural networks at various precision levels and explores how the compensation zone's effects vary by network structure, regardless of the nonsmooth functions' dimensionality (univariate or multivariate functions). We also analyze how nonsmooth MaxPool Jacobians affect neural network training's stability and performance.

Organization of the paper: In Section 2, we discuss the elements of nonsmooth backpropagation and define the subsets of network parameters - bifurcation, compensation, and regular zone. We also introduce nonsmooth MaxPool Jacobians and their theoretical implications for backpropagation, based on Bolte & Pauwels (2020a;b). In Section 3, we describe a numerical bifurcation and compensation zone with factors that influence their importance. In Section 4, we present detailed experiments on neural network training. Refer to Appendix C for further findings and experiments.

2 MaxPool neural networks and nonsmooth AD

2.1 Preliminaries and notations

In supervised training for neural networks, we work with a set of training data $(x_i, y_i)_{i=1}^N$, where each x_i is an input and y_i its matching label. A neural network, through its function f, uses parameters θ to generate predictions $\hat{y}_i = f(x_i, \theta)$. The difference between these predictions and the actual labels is measured by a loss function ℓ. The aim is to reduce this discrepancy across the training set by minimizing an empirical loss function L such as:

$$\min_{\theta \in \mathbb{R}^p} \quad L(\theta) := \frac{1}{N} \sum_{i=1}^{N} \ell(\hat{y}_i, y_i). \tag{1}$$

For all $i \in \{1, \dots, N\}$ and $\theta \in \mathbb{R}^p$, Equation (1) can be expressed with $\ell(\hat{y}_i, y_i) = l_i(\theta)$, where $l_i : \mathbb{R}^p \to \mathbb{R}$ represents a composition of M elementary functions as follows:

$$l_i(\theta) = g_{i,M} \circ g_{i,M-1} \circ \ldots \circ g_{i,1}(\theta). \tag{2}$$

Equation (2) models common neural network types, including feed-forward (Rumelhart et al., 1986a), convolutional (LeCun et al., 1998), and recurrent networks (Hochreiter & Schmidhuber, 1997). We focus on elementary functions that are locally Lipschitz and semialgebraic, commonly found in

[1] `https://github.com/AnonymousMaxPool/MaxPool-numerical`

nonsmooth neural networks (Bolte & Pauwels, 2020a;b). Functions $g_{i,j}$ include operations such as linear transformations, ReLU, MaxPool, convolution with filters, and softmax for classification.

2.2 Nonsmooth AD framework

Training nonsmooth neural networks (Bolte & Pauwels, 2020a; Bolte et al., 2021b; 2022; 2021a; Davis et al., 2020) is challenging due to the need to compute subgradients from Equation (1). Major machine learning tools such as TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and Jax (Bradbury et al., 2018) address this issue using automatic differentiation, referred to here as backprop (Rumelhart et al., 1986b; Baydin et al., 2018). They apply differential calculus to nonsmooth items, often replacing derivatives with *Clarke Jacobians* (Clarke, 1983). Given a locally Lipschitz continuous function $F : \mathbb{R}^p \to \mathbb{R}^q$, the *Clarke Jacobian* of F is defined as:

$$\operatorname{Jac}^{\mathrm{c}} F(x) = \operatorname{conv}\left\{ \lim_{k\to+\infty} \operatorname{Jac} F(x_k) : x_k \in \operatorname{diff}_F, x_k \underset{k\to+\infty}{\to} x \right\} \tag{3}$$

where diff_F represents the full measure set where F is differentiable and $\operatorname{Jac} F$ is the standard Jacobian of F. A selection v in $\operatorname{Jac}^{\mathrm{c}} F$ is a function $v\colon \mathbb{R}^p \to \mathbb{R}^{p\times q}$ such that, for all $x \in \mathbb{R}^p$, $v(x) \in \operatorname{Jac}^{\mathrm{c}} F(x)$. If F is C^1, the only possible selection is $v = \operatorname{Jac} F$.

Definition 1 (Calculus model, programs and nonsmooth AD) Let l be a composition function evaluated at $\theta \in \mathbb{R}^p$, as specified in Equation (2). A program P that executes l can be described through a sequence of subprograms such as:

- Elementary programs: $\{g_j\}_{j=1}^M$ such that $l(\theta) = g_M \circ g_{M-1} \circ \ldots \circ g_1(\theta)$.
- Derived programs: $\{v_j\}_{j=1}^M$ where each $v_j(w) \in \operatorname{Jac}^{\mathrm{c}} g_j(w)$ at point $w = g_{j-1} \circ \cdots \circ g_1(\theta)$.

Then, the backprop algorithm automates applying differential calculus rules as follows:

$$\operatorname{backprop}[P](\theta) = v_M\left(g_{M-1} \circ \ldots \circ g_1(\theta)\right) \cdot v_{M-1}\left(g_{M-2} \circ \ldots \circ g_1(\theta)\right) \cdot \ldots \cdot v_1(\theta). \tag{4}$$

In practice, AD libraries (Abadi et al., 2016; Paszke et al., 2019; Bradbury et al., 2018) implement dictionaries (see for e.g. Griewank & Walther (2008); Bolte et al. (2022)) containing conjointly elementary programs and derived programs which efficiently computes the quantities defined in Equation (4).

Remark 1 As seen in Section 1 with the zero program, various programs can implement a unique composition function l. Each elementary program g_j in the composition (see Definition 1) can be associated with different derived programs v_j. Specifically, for any $j = 1, \ldots, M$ and $w = g_{j-1} \circ \cdots \circ g_1(\theta)$, all selections $v_j(w)$ from the Clarke Jacobian of $g_j(w)$ can be used.

Example 1 The Clarke subdifferential of $\mathrm{ReLU}(t) = \max(0, t)$ at t is 0 for $t < 0$, 1 for $t > 0$, and the interval $[0, 1]$ for $t = 0$. All derived program that implements $\mathrm{ReLU}'(0) = s$ with $s \in [0, 1]$ can be used for backprop and have a variational bear.

Definition 2 (Backprop set) Let l denote a composition function evaluated at $\theta \in \mathbb{R}^p$, as specified in Equation (2). We define $J(\theta)$ as the function that encompasses the set of all possible backprop outputs through all programs implementing $l(\theta)$ as in Definition 1:

$$J(\theta) = \left\{ \operatorname{backprop}[P](\theta) : P \text{ is a program implementing } l(\theta) \right\}. \tag{5}$$

Remark 2 For a composition function l composed by C^1 elementary programs $\{g_j\}_{j=1}^M$, $J(\theta)$ is a singelton for all $\theta \in \mathbb{R}^p$. For locally Lipchitz semialgebraic (or definable) elementary programs $\{g_j\}_{j=1}^M$: Equation (4) is always an element within the backprop set.

Remark 3 The chain rule, essential for AD, often fails with Clarke subgradients. Hence, the backprop set might differ from the Clarke subdifferential (Clarke, 1983). For example, the Clarke subdifferential of $2\mathrm{ReLU}(x) - \frac{1}{3}\mathrm{ReLU}(-x)$ at $x = 0$ is $[\frac{1}{3}, 2]$, whereas backprop outputs 0 (with $\mathrm{ReLU}'(0) = 0$).

2.3 Network parameters subsets

Recently, Bertoin et al. (2023) analyzed the bifurcation zone for ReLU networks, which is defined by network parameters where AD's output diverges between $\mathrm{ReLU}'(0) = 0$ and $\mathrm{ReLU}'(0) = 1$. Their study, however, omitted the analysis of network parameters where backprop is theoretically a singleton, but AD inaccurately computes due to floating-point arithmetic, as shown in Table 1. To address this gap, we introduce the compensation zone.

Definition 3 (Compensation, bifurcation and regular zones) For each $i = 1, \ldots, N$, let l_i denote a composition function evaluated at $\theta \in \mathbb{R}^p$ and $J_i(\theta)$ denote the backprop set associated as detailed in Definition 2. We define the following network parameters subsets of $\mathbb{R}^p$:

$$\Theta_R = \left\{\theta \in \mathbb{R}^P : \forall i, j \in \{1, \ldots, N\} \times \{1, \ldots, M\}, \mathrm{Jac}^{\mathrm{c}}\, g_{i,j}(w) \text{ is a singleton}\right\}, \tag{6}$$

$$\Theta_C = \left\{\theta \in \mathbb{R}^P \backslash \Theta_R : \forall i \in \{1, \ldots, N\}, J_i(\theta) \text{ is a singleton}\right\}, \tag{7}$$

$$\Theta_B = \left\{\theta \in \mathbb{R}^p \backslash \Theta_R : \exists i \in \{1, \ldots, N\} \text{ such that } J_i(\theta) \text{ is not a singleton}\right\}, \tag{8}$$

where $w = g_{i,j-1} \circ \ldots \circ g_{i,1}(\theta)$, Θ_R is the regular zone, Θ_C the compensation zone and Θ_B the bifurcation zone.

The mathematical tools of Proposition 1 are conservative fields developed in Bolte & Pauwels (2020a). This proposition implies that theoretically (assuming exact arithmetic over the reals), the backprop set is almost everywhere a singleton. The proof is given in Appendix B.

Proposition 1 Given subsets Θ_R, Θ_B, and Θ_C in $\mathbb{R}^p$ as defined in Definition 3, the following properties hold:

- Θ_R, Θ_B, and Θ_C form a partition of $\mathbb{R}^p$.
- Θ_B is a Lebesgue null measure subset.

Remark 4 (Backprop returns a gradient a.e.) Let $\theta \in \mathbb{R}^p$ and P be a program implementing a composition function $l(\theta)$ as in Definition 1. Then $\mathrm{backprop}[P](\theta) = \nabla l(\theta)$ almost everywhere.

2.4 MaxPool-derived programs

Definition 4 (Clarke Jacobian of matrix's maximum function) Let X be a $m \times n$ real matrix and F_s be a function such that $F_s(X) = \max_{1 \le i \le m, 1 \le j \le n} X_{ij} \in \mathbb{R}$, where $s := m \times n$ denotes the size of X. The Clarke Jacobian of F_s at the point X is:

$$\mathrm{Jac}^{\mathrm{c}}\, F_s(X) = \mathrm{conv}\left(\bigcup_{(i,j) \in A(X)} E_{ij}\right), \tag{9}$$

where $A(X) := \{(i,j) \in \{1,\dots,m\} \times \{1,\dots,n\} : F_s(X) = X_{ij}\}$ is the active set and E_{ij} is an $m \times n$ matrix with all entries equal to 0 except for the (i,j)-th entry which is 1.

Definition 5 (MaxPool operation) Let $X \in \mathbb{R}^{p \times q}$ be a real matrix, and $s := m \times n$ be the size of a pooling window such that $p \geq m$ and $q \geq n$. For each $i \in \{0,\dots,\lfloor \frac{p}{m} \rfloor - 1\}$ and $j \in \{0,\dots,\lfloor \frac{q}{n} \rfloor - 1\}$, we define a submatrix $X_{i,j}$ of X, of size $m \times n$ as follows:

$$X_{i,j} := \{X_{kl} : m \times i \leq k < m \times (i+1), n \times j \leq l < n \times (j+1)\}, \tag{10}$$

where k and l are the indices of the entries in X, in the lexicographic order. The MaxPool operation output a matrix $Y \in \mathbb{R}^{\lfloor \frac{p}{m} \rfloor \times \lfloor \frac{q}{n} \rfloor}$ where $Y_{ij} = F_s(X_{i,j})$ for all $i \in \{0,\dots,\lfloor \frac{p}{m} \rfloor - 1\}$ and $j \in \{0,\dots,\lfloor \frac{q}{n} \rfloor - 1\}$. Finally, the MaxPool Clarke Jacobian at point X, denoted as $\mathrm{Jac}^{\mathrm{c}}\,\mathrm{MaxPool}(X)$, can be obtained by replacing each submatrix $X_{i,j}$ in X with $\mathrm{Jac}^{\mathrm{c}}\,F_s(X_{i,j})$.

Definition 6 (MaxPool-derived programs) Define $X_{i,j} \in \mathbb{R}^{m \times n}$ as a submatrix of X (Definition 5), from which we derive MaxPool programs based on the Clarke Jacobian:

- **Native:** Chooses the first index (i_1, j_1) from the active set $A(X_{i,j})$ and outputs $E_{i_1 j_1}$. Autograd libraries use this implementation.
- **Minimal:** Takes all indices from $A(X_{i,j})$, averaging them as $\frac{1}{|A(X_{i,j})|} \sum_{(k,l) \in A(X_{i,j})} E_{kl}$. We called it "minimal" as it yields the smallest norm element within Equation (9).
- **Hybrid:** A blend of native and minimal, parameterized by $\beta > 0$:

$$(1-\beta) \times E_{i_1 j_1} + \beta \times \left(\frac{1}{|A(X_{i,j})|} \sum_{(k,l) \in A(X_{i,j})} E_{kl} \right),$$

Remark 5 The hybrid MaxPool-derived program is a selection of the MaxPool Clarke Jacobian for $\beta \in [0,1]$ and a selection of a conservative Jacobian approach for other β values, as outlined in Bolte & Pauwels (2020a) .

3 Numerical AD with MaxPool-derived programs

In this section, we explore subsets of network parameters for neural networks with MaxPool operations at various floating-point precisions. We note that the numerical bifurcation zone identified by Bertoin et al. (2023) is not applicable to our MaxPool-derived program analysis. As indicated in Table 1, our investigation extends to include small AD errors that occur with floating-point but not with real numbers. Therefore, we explore both numerical bifurcation and compensation zones via numerical methods, employing notations from Sections 2.1 and 2.2.

3.1 A numerical criteria for the bifurcation and compensation zone

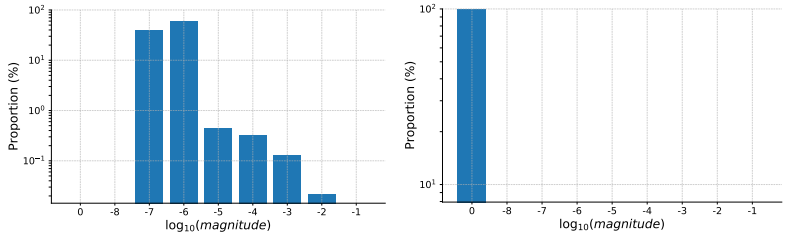
Recently, Bertoin et al. (2023) studied a numerical bifurcation zone S_{01} for ReLU-based programs. For each $i = 1,\dots,N$, two versions of a program, R_i^0 (using $\mathrm{ReLU}'(0) = 0$) and R_i^1 (using $\mathrm{ReLU}'(0) = 1$), implement a function l_i. The bifurcation zone S_{01} is defined as:

$$S_{01} = \left\{ \theta \in \mathbb{R}^P : \ \exists i \in \{1,\dots,N\}, \mathrm{backprop}[R_i^0](\theta) \neq \mathrm{backprop}[R_i^1](\theta) \right\}. \tag{11}$$

Definition 7 (Backprop variation) Let $(B_q)_{q\in\mathbb{N}}$ be a sequence of mini-batches, where each batch size $|B_q|$ falls within $\{1,\ldots,N\}$. Consider $P=\{P_i\}_{i=1}^N$ and $Q=\{Q_i\}_{i=1}^N$ as two neural network implementations using different MaxPool-derived programs (e.g., native vs. minimal). Each P_i and Q_i applies a composition function l_i. The backprop variation between P and Q over M experiments with random parameters $\{\theta_m\}_{m=1}^M$ is defined as:

$$D_{m,q}(P,Q)=\left\|\text{backprop}\left[\sum_{i\in B_q}P_i(\theta_m)\right]-\text{backprop}\left[\sum_{i\in B_q}Q_i(\theta_m)\right]\right\|_1. \tag{12}$$

A 32 bits MNIST experiment: Let P and Q be programs for a LeNet-5 network on MNIST, using native and minimal MaxPool programs, respectively. For a sanity check, let $\tilde{P}$ be a copy of P. We compute the backprop variation (see Definition 7) between P and $\tilde{P}$ and between P and Q. We control all sources of divergence in our implementation using deterministic computation. Results are reported in Figure 1 and the experiment was run on a CPU under 32 bits precision.



(a) $D_{m,q}(P,Q)$ (b) $D_{m,q}(P,\tilde{P})$

Figure 1: Histogram of backprop variation $D_{m,q}$ for LeNet-5 on MNIST (128 mini-batch size) at 32-bit precision, comparing P with $\tilde{P}$ and P with Q over $M=1000$ experiments.

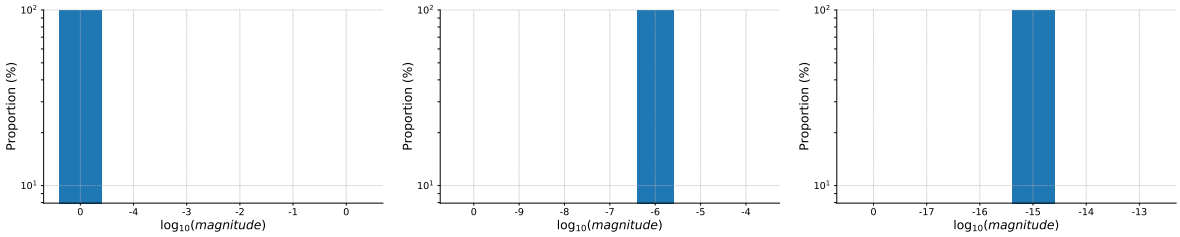
We find no backprop variation between P and $\tilde{P}$, indicating controlled divergence sources. Contrary to expectations (Proposition 1) of no variation between P and Q, $D_{m,q}(P,Q)>0$ across all m,q. We identify two variation types: minor ones (98.78% of parameters) around the value of machine precision in 32 bits (10^{-8} and 10^{-7}) and major ones up to 10^{-3} (1.22% of parameters). This contrasts with Bertoin et al. (2023) experiments, which reported significant divergences or none.

An heuristic for the numerical bifurcation zone: In Figure 1, two backprop variation types are noted: one potentially from numerical bifurcation and another due to floating-point arithmetic (compensation errors). To analyze these, we compare observed backprop variations with known sources: GPU nondeterminism (see Appendix A.6.1) and variations from ReLU-derived programs in 16 and 32-bit precision (Bertoin et al., 2023). This approach distinguishes between numerical bifurcation and compensation without assuming separate zones. Let ω denote floating-point precision and f a neural network like LeNet-5, VGG, or ResNet.

A threshold with nondeterministic GPU calculations: We set a threshold $\tau^1_{f,\omega}$ for the maximum backprop variation due to nondeterministic GPU calculations (see Appendix A.6.1):

$$\tau^1_{f,\omega}=\max_{1\le m\le M,1\le q}D_{m,q}(P,\tilde{P}) \tag{13}$$

where P and $\tilde{P}$ implement a neural network f using the same MaxPool derived-program. Refer to Figure 2 for an illustration. No variation is noted at $\omega=16$, with PyTorch disabled nondeterministic GPU operations.



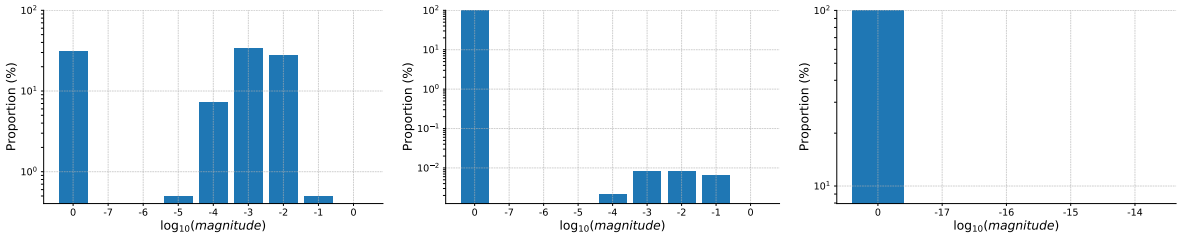
(a) $\tau^1_{f,16} = 0$ (b) $\tau^1_{f,32} = 1.11 \times 10^{-7}$ (c) $\tau^1_{f,64} = 1.55 \times 10^{-16}$

Figure 2: Histogram of backprop variation under nondeterministic GPU operations, where f is a LeNet-5 network on MNIST with batch size 128 for $K = 1000$ experiments.

A threshold with ReLU**-derived programs:** For ReLU-derived programs, we define R^0 (with $\text{ReLU}'(0) = 0$) and R^1 (with $\text{ReLU}'(0) = 1$) as two implementations of network f. We introduce threshold $\tau^2_{f,\omega}$ for backprop variation:

$$\tau^2_{f,\omega} = \min_{1 \le m \le M, 1 \le q} \left\{ D_{m,q}(R^0, R^1) : D_{m,q}(R^0, R^1) > 0 \right\}, \tag{14}$$

as shown in Figure 3, ensuring deterministic GPU operations.



(a) $\tau^2_{f,16} = 3.39 \times 10^{-6}$ (b) $\tau^2_{f,32} = 4.81 \times 10^{-5}$ (c) $\tau^2_{f,64} = 0$

Figure 3: Histogram of backprop variation with ReLU-derived programs, where f is a LeNet-5 network on MNIST with batch size 128, $K = 1000$ experiments.

Figure 3 shows two backprop variation types: significant divergences or none, similar to Bertoin et al. (2023)'s findings. These divergences might indicate a numerical bifurcation zone. Variations from nondeterministic GPU calculations, shown in Figure 2, align with minor variations near machine precision seen in Figure 1. We suggest a numerical bifurcation zone, using different thresholds for various precisions due to hardware constraints.

Criteria 1 (Numerical bifurcation zone) For a neural network f and a floating-point precision ω, let $\tau_{f,\omega}$ be a fixed threshold (for e.g $\tau^1_{f,\omega}$, $\tau^2_{f,\omega}$). The numerical bifurcation zone is defined as:

$$S(\tau_{f,\omega}) = \left\{ \theta \in \mathbb{R}^P : \exists i \in \{1, \ldots, N\}, \|\text{backprop}[P_i](\theta) - \text{backprop}[Q_i](\theta)\|_1 > \tau_{f,\omega} \right\} \subset \Theta_B. \tag{15}$$

Here, P_i and Q_i represent programs for f using different MaxPool-derived programs.

Table 4 in Appendix A.6 lists threshold values for different networks and datasets at 16-bit, 32-bit, and 64-bit precisions. These thresholds are numerical guides and vary with initial network parameters, datasets, and architecture. The existence and characteristics of the compensation zone

are determined by the neural network's structure, not by the nature (univariate or multivariate) of the elementary programs in Definition 1. Specifically, ReLU-based approaches can induce compensation errors as shown in Table 1 (see Appendix A.3). For convolutional networks like VGG or ResNet, substituting MaxPool with ReLU functions, according to the formula $2\max(x, y) = (x+y)+(\mathrm{ReLU}(x)-\mathrm{ReLU}(-y))+(\mathrm{ReLU}(y)-\mathrm{ReLU}(-x))$, modifies the bifurcation zone phenomenon identified in Bertoin et al. (2023). Conversely, using NormPool—a nonsmooth multivariate function calculating the Euclidean norm—avoids such compensation errors. Refer to Appendix A.4 for more details.

3.2 Volume of the numerical bifurcation zone

We used Monte Carlo sampling to estimate the numerical bifurcation zone's volume for various networks, following Criteria 1 and detailed in Appendix A.6.3. Thresholds $\tau^2_{f,16}$, $\tau^1_{f,32}$, and $\tau^1_{f,64}$ were applied across all networks, as specified in Equations (13) and (14).

Experimental setup: We generate a set of network parameters $\{\theta_m\}_{m=1}^M$ randomly using Kaiming-Uniform initialization (He et al., 2015), with $M = 1000$. Then, we iterate over the entire CIFAR10 dataset to estimate the proportion of θ_m in the numerical bifurcation zone S defined in Criteria 1 (as shown in Equation equation 18) and the proportion of impacted mini-batches (as shown in Equation equation 19).

Impact of floating-point precision: Using VGG11 on CIFAR10, we assessed S's volume for different precisions. Results indicate that at 16-bit and 32-bit precision, all parameters fell within S, while at 64-bit, none did. The impact on mini-batches was 46% at 32 bits and 100% at 16 bits, highlighting precision's role in backprop effects with MaxPool-derived programs.

Floating-point precision	16 bits	32 bits	64 bits
Proportion of $\{\theta_m\}_{m=1}^M$ in S	100%	100%	0%
Proportion of impacted mini-batches	100%	46.67%	0%

Table 2: Impact of S according to floating-point precision using a VGG11, on CIFAR10 dataset and $M = 1000$ experiments. The first line represents network parameters θ_m in S, while the second measured the proportion of affected mini-batches falling in S.

We examined the effect of mini-batch size on the proportion of affected mini-batches in S using VGG11 on CIFAR10. Larger mini-batch sizes increase the proportion impacted at 32-bit precision, but no parameters fell into S at 64-bit precision (Figure 4). Network depth (VGG variants 11, 13, 16, 19) did not significantly change the impact on mini-batches at 16-bit and 32-bit precisions. However, batch normalization significantly increased the affected mini-batches at 32-bit precision.

4 Impact on learning

4.1 Benchmarks and implementation

Datasets and architectures: We train neural networks to investigate the impact of numerical effects outlined in Section 3. Our experiments used CIFAR10 (Krizhevsky & Hinton, 2010), MNIST (LeCun et al., 1998) and ImageNet (Deng et al., 2009) datasets. We test various network architectures

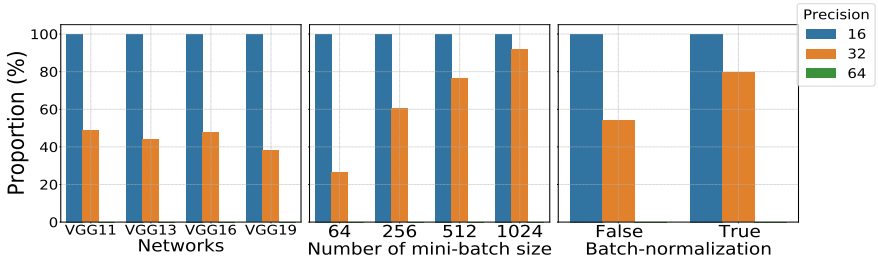


Figure 4: Impact of different size parameters on the proportion of affected mini-batches (Equation equation 19) using CIFAR10 dataset. First: Different VGG network sizes. Second: VGG11 with varying mini-batch sizes. Third: VGG11 with and without batch normalization.

including VGG11 (Simonyan & Zisserman, 2014), ResNet (He et al., 2016), and LeNet (LeCun et al., 1998). Details are available in Appendix C.1.

Training settings: The default optimizer is SGD. Conducted on PyTorch and Nvidia V100 GPUs, we define mini-batch sequences $(B_k)_{k\in\mathbb{N}}$ with sizes $|B_k| \subset \{1,\ldots,N\}$, where $\alpha_k > 0$ is the learning rate for each mini-batch k. Each program P_i in $P = \{P_i\}_{i=1}^{N}$ implements a function l_i (as in Definition 1). The SGD algorithm updates network parameters $\theta_{k,P}$ by:

$$\theta_{k+1,P} = \theta_{k,P} - \gamma \frac{\alpha_k}{|B_k|} \sum_{i\in B_k} \text{backprop}[P_i](\theta_{k,P}) \tag{16}$$

with $\gamma > 0$ indicating the step-size parameter.

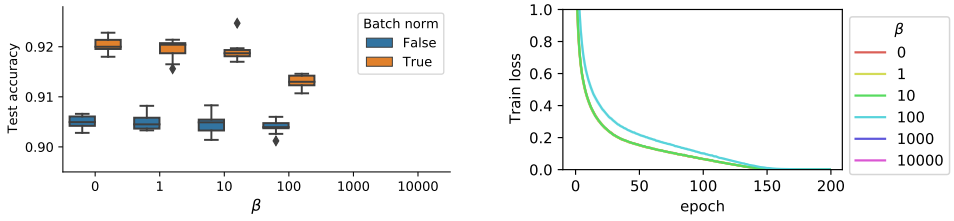
4.2 Effect on training and test errors

We trained a VGG11 on CIFAR10 using SGD, testing hybrid MaxPool programs across 16 and 32-bit precisions with varied β values, repeating each setup ten times with random initializations. Results in Figure 5 align with the revised findings from Bertoin et al. (2023), unlike the initial findings. Across architectures and datasets, our results were consistent, yet β sensitivity varied. Large β values could destabilize training and lower test accuracy, with $\beta = 0$ generally effective but not always optimal. Figure 5 shows effects for $\beta \in 1, 10, 100$. Using the Adam optimizer with 32-bit precision, as noted by Bertoin et al. (2023), mitigates large β effects, stabilizing training (Appendix C.2).

Training effect with 16-bit: For β values greater than 10^3, we observe training instability and exploding gradients, regardless of batch normalization. Stable and efficient test accuracy persists for $\beta \in \{0, 1, 10, 100\}$.

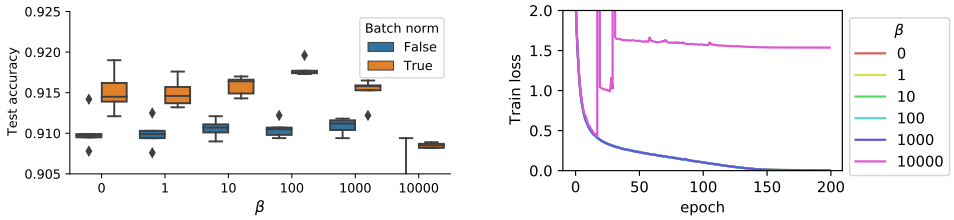
Training effect with 32-bit: When the value of β is large (e.g. 10^4), training can become unstable, leading to oscillations and sudden jumps in the learning process if batch normalization is not applied. However, using batch normalization with a large β value can prevent this issue, resulting in improved accuracy on test data and avoiding the problem of gradient explosion.

Training and weight differences: We trained seven VGG11 networks $\{P_i\}_{i=0}^{6}$ at 32-bit precision on CIFAR10 for 200 epochs, using 128-size mini-batches, fixed learning rate $\alpha_k = 1$, and momentum $\gamma \in [0.01, 0.012]$. All networks, starting with the same parameters, varied in hybrid MaxPool $\{\beta_i\}_{i=0}^{6}$.



(a) 16-bit

(b) 16-bit, with batch normalization



(c) 32-bit

(d) 32-bit, without batch normalization

Figure 5: Training a VGG network on CIFAR10 with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).

With nondeterministic GPU computation, we measured epoch-wise backpropagation differences between P_0 and the others, observing parameter variations and test accuracies. Variations and accuracies for $\beta \leq 10^3$ were consistent, showing β's minimal impact. At $\beta = 10^4$, significant divergences and a test accuracy drop were noted, indicating that high β values could destabilize training due to exploding gradients.

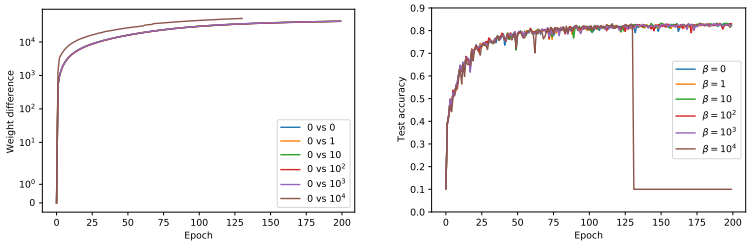


Figure 6: Left: Difference between network parameters (L^1 norm) at each epoch. "0 vs 0" indicates $\|\theta_{k,P_0} - \theta_{k,P_7}\|_1$ where P_7 is a second run of P_0 for sanity check, "0 vs 1" indicates $\|\theta_{k,P_0} - \theta_{k,P_1}\|_1$. Right: test accuracy of each $\{P_i\}_{i=0}^5$ during 200 epochs.

5 Conclusion

In our study, we assess autodiff reliability in neural networks employing MaxPool. Testing across various models and datasets, we found AD might inaccurately handle MaxPool with floating-point calculations. This suggests Lee et al. (2023)'s AD correctness findings may not fully extend to convolutional neural networks using MaxPool. Our analysis focuses on two subsets: bifurcation zones, where AD inaccuracies occur in both real and floating-point calculations, and compensation zones, correct in real numbers but possibly erroneous in floating-point numbers. Bifurcation zones,

though rare, lead to notable AD divergences, while compensation zones more commonly show minor, machine precision-related amplitude shifts.

Lower-norm MaxPool Jacobians enhance training stability and test accuracy, whereas higher-norm Jacobians risk training instability, particularly in lower-precision settings. Factors like dataset, architecture, and learning parameters such as batch normalization and the Adam optimizer significantly impact AD's numerical behavior.

Acknowledgments and Disclosure of Funding

The author acknowledges the support of the AI Interdisciplinary Institute ANITI funding under the grant agreement ANR-19-PI3A-0004. The author acknowledges the help of the Association Nationale de la Recherche et de la Technologie (ANRT) and Thales LAS France, which contributed to Ryan B's grant. This work was performed using HPC resources from CALMIP (Grant 2023-[P23040]). The author would like to thank Jérôme Bolte and Edouard Pauwels for their precious feedback. The author would like to thank the collaborators at Thales LAS France, in particular Beatrice Pesquet-Popescu and Andrei Purica, for their help. comments.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pp. 265–283, 2016. URL `https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf`.

Paul I. Barton, Kamil A. Khan, Peter Stechlinski, and Harry A.J. Watson. Computationally relevant generalized derivatives: theory, evaluation and applications. *Optimization Methods and Software*, 33(4-6):1030–1072, 2018. doi: 10.1080/10556788.2017.1374385. URL `https://doi.org/10.1080/10556788.2017.1374385`.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

David Bertoin, Jérôme Bolte, Sébastien Gerchinovitz, and Edouard Pauwels. Numerical influence of relu'(0) on backpropagation, 2023.

Jérôme Bolte and Edouard Pauwels. Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning. *Mathematical Programming*, pp. 1–33, 2020a.

Jérôme Bolte, Tam Le, Edouard Pauwels, and Antonio Silveti-Falls. Nonsmooth implicit differentiation for machine learning and optimization. *CoRR*, abs/2106.04350, 2021a. URL `https://arxiv.org/abs/2106.04350`.

Jérôme Bolte, Tam Le, Edouard Pauwels, and Tony Silveti-Falls. Nonsmooth implicit differentiation for machine-learning and optimization. *Advances in Neural Information Processing Systems*, 34, 2021b.

Jérôme Bolte, Ryan Boustany, Edouard Pauwels, and Béatrice Pesquet-Popescu. On the complexity of nonsmooth automatic differentiation. In *The Eleventh International Conference on Learning Representations*, 2022.

Jérôme Bolte and Edouard Pauwels. A mathematical model for automatic differentiation in machine learning. In *Conference on Neural Information Processing Systems*, 2020b.

Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311, 2018.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL `http://github.com/google/jax`.

Frank H Clarke. *Optimization and nonsmooth analysis*. SIAM, 1983.

Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. In *Proceedings of the International Conference on Learning Representations*, 2015.

D. Davis, D. Drusvyatskiy, S. Kakade, and J. D. Lee. Stochastic subgradient method converges on tame functions. *Foundations of Computational Mathematics.*, 2018.

Damek Davis, Dmitriy Drusvyatskiy, Sham Kakade, and Jason D Lee. Stochastic subgradient method converges on tame functions. *Foundations of computational mathematics*, 20(1):119–154, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, 2009.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

A. Griewank and A. Rojas. Treating artificial neural net training as a nonsmooth global optimization problem. *In International Conference on Machine Learning, Optimization, and Data Science (pp. 759-770). Springer, Cham.*, 2019.

A. Griewank and A. Walther. Beyond the oracle: Opportunities of piecewise differentiation. *In Numerical Nonsmooth Optimization (pp. 331-361). Springer, Cham.*, 2020.

Andreas Griewank. On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods and Software*, 28, 07 2013. doi: 10.1080/10556788.2013.796683.

Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

Andreas Griewank, Andrea Walther, Sabrina Fiege, and Torsten Bosse. On lipschitz optimization based on gray-box piecewise linearization. *Mathematical Programming*, 158:383–415, 2016.

Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pp. 1737–1746. PMLR, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9: 1735–1780, 1997.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 4700–4708, 2017.

Kyuyeon Hwang and Wonyong Sung. Fixed-point quantization of deep convolutional networks. In *Proceedings of the International Conference on Machine Learning*, 2014.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, et al. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. *arXiv preprint arXiv:1807.11205*, 2018.

Sham M Kakade and Jason D Lee. Provably correct automatic sub-differentiation for qualified programs. In *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Alex Krizhevsky and Geoff Hinton. Convolutional deep belief networks on cifar-10. *Unpublished manuscript*, 40(7):1–9, 2010.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pp. 1097–1105, 2012.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. On correctness of automatic differentiation for non-differentiable functions. In *NeurIPS 2020-34th Conference on Neural Information Processing Systems*, 2020.

Wonyeol Lee, Sejun Park, and Alex Aiken. On the correctness of automatic differentiation for neural networks with machine-representable parameters, 2023.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, pp. 5, 2011.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

D. E. Rumelhart, Geoffrey E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986a.

David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986b.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Bert Speelpenning. *Compiling fast partial derivatives of functions given by algorithms*. University of Illinois at Urbana-Champaign, 1980.

Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proceedings of the Deep Learning and Unsupervised Feature Learning Workshop*, 2011.

Kouichi Yamaguchi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. A neural network for speaker-independent isolated word recognition. In *ICSLP*, 1990.

Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pp. 818–833, 2014.

This is the appendix for "On the numerical reliability of nonsmooth autodiff: a MaxPool case study".

Contents

A Further comments, discussion, and technical elements

A.1 Implementation of the zero program

The implementation of the zero function used in Table 1 is given in Figure 7. Programs $\max_1$ and $\max_2$ correspond to an equivalent implementation of the same function max, but the computed derivatives are different.

```
def max1(x):
    # Derivative: first coordinate
    # Not the default in Torch
    res = x[0]
    for i in range(1, 4):
        if x[i] > res:
            res = x[i]
    return res
```

```
def max2(x):
    # Derivative: min norm
    # Default in Jax
    return torch.max(x)

def zero(t):
    # Zero function
    z = t * x
    return max1(z) - max2(z)
```

Figure 7: Implementation of programs $\max_1$, $\max_2$ and zero using Pytorch. Programs $\max_1$ and $\max_2$ are an equivalent implementation of max, but with different derivatives due to the implementation.

A.2 Challenges posed by MaxPool in image processing

In Convolutional Neural Networks (CNNs), the MaxPool operation is frequently used for reducing dimensions and downsampling. This function is especially crucial in image contexts, where uniform intensity regions are common, especially around the edges of objects and flat surfaces. One common situation is encountering identical pixel values within a pooling window, as shown in Figure 8. MaxPool must choose among these equivalent values, creating a point of non-differentiability. During training, this affects gradient calculation in backpropagation, affecting the updates to convolutional filters (Goodfellow et al., 2016).

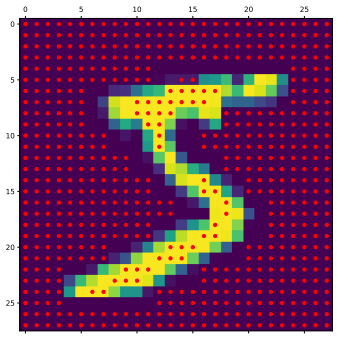


Figure 8: Image segment post-convolution, spotlighting equal pixel values (marked in red) within a 2x2 MaxPool window.

A.3 AD errors with ReLU-derived programs

We conduct a small PyTorch experiment using the nonsmooth function $\mathrm{ReLU}\colon x \mapsto \max(x, 0)$. Consider two programs $\max_1$ and $\max_2$ implementing the $\max\colon x \mapsto \max_{1 \leq i \leq 4} x_i \in \mathbb{R}$ function using different ReLU-derived programs. Note that $2\max(x, y) = (x + y) + (\mathrm{ReLU}(x) - \mathrm{ReLU}(-y)) + (\mathrm{ReLU}(y) - \mathrm{ReLU}(-x))$. Let $\mathrm{zero}_2\colon t \mapsto \max_1(t \times x) - \max_2(t \times x)$ be a program implementing the null function as described in Figure 9. Let zero_2' denote the backward AD algorithm for the zero program. As mathematical functions, $\max_1$ and $\max_2$ are equal and the program zero outputs constantly 0. However, for some $t \in \mathbb{R}$, AD can return $\mathrm{zero}_2'(t) \neq 0$. Results are reported in Table 3 and similar to Table 1.

```
def relu(x):
    return torch.relu(x)

def relu2(x):
    return torch.where(x >= 0, x, torch.tensor(0.0))

def max01(x):
    return (x[0] + x[1]) / 2 + relu((x[0] - x[1]) / 2) + relu((x[1] - x[0]) / 2)

def max02(x):
    return (x[0] + x[1]) / 2 + relu2((x[0] - x[1]) / 2) + relu((x[1] - x[0]) / 2)

def max1(x):
    return max01(torch.stack([max01(x[0:2]), max01(x[2:4])]))

def max2(x):
    return max02(torch.stack([max02(x[0:2]), max02(x[2:4])]))

def zero_2(t):
    z = t * x
    return max1(z) - max2(z)
```

Figure 9: Implementation of $\max_1$, $\max_2$ and zero_2 using Pytorch. Programs $\max_1$ and $\max_2$ are an equivalent implementation of max, but implemented using different ReLU-derived programs.

					$\text{zero}_2'(t)$						
t					-10^{-3}	-10^{-2}	-10^{-1}	0	10^{1}	10^{2}	10^{3}
$x =$	1.0	2.0	3.0	4.0	0.0	0.0	0.0	1.5	0.0	0.0	0.0
$x =$	1.4	1.4	1.4	1.4	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}	10^{-7}

Table 3: Summary of various types of AD errors with zero_2 program using PyTorch for different combinations of t and x.

A.4 NormPool : a nonsmooth multivariate operation without compensation errors

We conducted an experiment to show that compensation errors are not caused by the multivariate nature of nonsmooth elementary functions when using floating-point arithmetic. In this experiment, we used the NormPool operation, which is similar to the MaxPool operation but replaces the maximum with the Euclidian norm. Two programs, P and Q, were used to implement a LeNet-5 network on the MNIST dataset with two different NormPool-derived programs. We computed the backprop variation (see Definition 7) between P and Q, while controlling all sources of divergence in our implementation using deterministic computation. The results are presented in Figure 10. The experiment was conducted on a CPU with 16-bit floating-point precision.

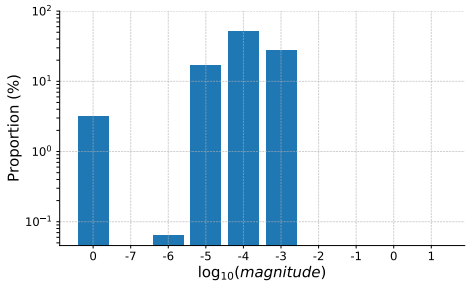


Figure 10: Histogram of backprop variation between P and Q for a LeNet-5 network on MNIST (128 mini-batch size) with 16-bit. We run $M = 1000$ experiments.

In contrast to our findings with MaxPool, we obtained similar results to those reported in Bertoin et al. (2023) with ReLU-based programs. Specifically, for NormPool-based programs, we observed either significant divergence of backprop or none.

A.5 Bifurcation zone: a practical example

This section presents an example that demonstrates cases where AD can be incorrect. Calculating the accurate derivative for all inputs might be impossible, particularly when the function is nondifferentiable. This is because the derivative does not exist for inputs where the function is nondifferentiable.

A.5.1 Network configuration

Consider an input matrix X of size 4×4 given by:

$$X = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \qquad \text{(Input)}$$

Let k be a positive number and W be a convolution kernel of size 3×3 given by:

$$W = k \cdot \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \qquad \text{(Convolution kernel)}$$

Let's consider a composition function l such that:

$$l(W) = \text{MaxPool} \circ (X * W) = k \tag{17}$$

where the convolution operation $X * W$ produces an output matrix Z of size 2×2, followed by the application of a MaxPool with a pooling window of size 2×2.

A.5.2 Backprop computation: native vs minimal

Let P (resp. Q) be a program implementing the composition function l in Equation equation 17 using the native (resp. minimal) MaxPool-derived program. Then, we have:

$$\text{backprop}[P](W) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad \text{backprop}[Q](W) = \begin{pmatrix} 0.5 & 0 & 0.5 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

The convolutional kernel W falls within the bifurcation zone defined in Definition 3.

A.6 Comments on Section 3

A.6.1 Non-determinism in GPU computation

Graphics Processing Units (GPUs) are designed for parallel processing, which can result in unpredictable behaviors.

- **Floating-point operations:** The non-associative nature of floating-point arithmetic can lead to discrepancies. These differences might become significant as they accumulate across operations.
- **Reduction operations:** Functions like sum or maximum, especially in GPUs, can exhibit variability between runs. This variability can result in divergent accumulated rounding errors.

A.6.2 Threshold values for various networks in Section 3.1

Table 4 presents threshold values for various neural networks on different datasets, computed under different floating-point precisions (16-bit, 32-bit, and 64-bit). For simplicity, thresholds are approximated as powers of 10.

A.6.3 Details on Monte Carlo sampling in Section 3.2

Recall that, for a neural network f and a floating-point precision ω, we want to estimate the volume of the set

$$S(\tau_{f,\omega}) = \left\{\theta \in \mathbb{R}^P : \exists i \in \{1, \ldots, N\}, \|\text{backprop}[P_i](\theta) - \text{backprop}[Q_i](\theta)\|_1 > \tau_{f,\omega}\right\} \subset \Theta_B$$

Network f	**Dataset**	$\tau^1_{f,16}$	$\tau^2_{f,16}$	$\tau^1_{f,32}$	$\tau^2_{f,32}$	$\tau^1_{f,64}$	$\tau^2_{f,64}$
LeNet-5	MNIST	0	10^{-5}	10^{-6}	10^{-5}	10^{-14}	0
VGG-11	CIFAR-10	0	10^{-1}	10^{-8}	10^{-7}	10^{-14}	0
VGG-11	SVHN	0	10^{-1}	10^{-8}	10^{-7}	10^{-15}	0
VGG-13	CIFAR-10	0	10^{-1}	10^{-9}	10^{-9}	10^{-14}	0
VGG-16	CIFAR-10	0	10^{-2}	10^{-10}	10^{-9}	10^{-15}	0
VGG-19	CIFAR-10	0	10^{-3}	10^{-11}	10^{-10}	10^{-15}	0
ResNet-18	CIFAR-10	10^{-2}	1	10^{-3}	10^{-4}	10^{-13}	0
DenseNet-121	CIFAR-100	0	10^{-2}	10^{-6}	10^{-1}	10^{-14}	0

Table 4: Threshold values of various neural networks f across different datasets.

Our experiments divide a dataset into R mini-batches. Each r-th mini-batch is represented by the index set $B_r \subset \{1, \ldots, N\}$. The programs P_r and Q_r are associated with the neural network f and implement a composition function l_r for each r. Specifically, P_r uses the native MaxPool-derived program, whereas Q_r uses the minimal one. For every precision level $\omega \in \{16, 32, 64\}$, we establish a threshold $\tau_{f,\omega}$ as in Section 3. Using the Kaiming-Uniform (He et al., 2015) initialization in PyTorch, we randomly generate a parameter set $\{\theta_j\}_{j=1}^K$, with $K = 1000$. The first line of Table 2 is given by the formula

$$\frac{1}{K}\sum_{k=1}^{K} \mathbb{1}\!\!\!\!\mathbb{1}\left(\exists r \in \{1,\ldots,R\}, \left\|\text{backprop}\left[\sum_{j\in B_r} P_j(\theta)\right] - \text{backprop}\left[\sum_{j\in B_r} Q_j(\theta)\right]\right\|_1 > \tau_{f,\omega}\right), \quad (18)$$

where $\mathbb{1}\!\!\!\!\mathbb{1}$ represents the indicator function, returning either 1 or 0 depending on the truth value of its argument's condition. Similarly, the second line of Table 2 is given by the formula

$$\frac{1}{KR}\sum_{k=1}^{K}\sum_{r=1}^{R} \mathbb{1}\!\!\!\!\mathbb{1}\left(\left\|\text{backprop}\left[\sum_{j\in B_r} P_j(\theta)\right] - \text{backprop}\left[\sum_{j\in B_r} Q_j(\theta)\right]\right\|_1 > \tau_{f,\omega}\right), \quad (19)$$

Using the formula

$$\sqrt{\frac{\ln\left(\frac{2}{\alpha}\right)}{2n}},$$

and setting $\alpha = 0.05$, we compute the error margin of the Hoeffding confidence interval as $n = K$ for Table 2's first line and $n = KR$ for its second. The first line adheres to a 95% confidence interval under the *iid* assumption due to Hoeffding's inequality.

Using McDiarmid's inequality at risk level $\alpha = 0.05$, we compute the error margin of the second line in Table 2 by the formula

$$\sqrt{\frac{1}{2}\left(\frac{1}{K}+\frac{1}{R}\right)\ln\left(\frac{2}{\alpha}\right)}.$$

B Proof related to Section 2.3

Proof 1 (of Proposition 1)

1. The three subsets have unique definitions, indicating that they are separate. For instance, a parameter cannot belong to the regular and bifurcation zones since the regular zone is defined as the area where each program $g_{i,j}$ is assessed at differentiable points. On the other hand, the bifurcation zone is defined as the region where the set of all possible backprop outputs is not a singleton, indicating non-differentiability at some points. Additionally, the union of these zones covers the entire parameter space Θ as every parameter must be assigned to one of the three subsets: resulting in differentiable points when evaluated, resulting in nondifferentiable points but having a singleton backprop set, or resulting in nondifferentiable points with a non-singleton backprop set. Therefore, $\Theta_R \cup \Theta_B \cup \Theta_C = \Theta$.

2. As we consider locally Lipchitz semialgebraic (or definable) functions, see [Theorem 1, Bolte & Pauwels (2020a)] for the proof arguments.

C Complements on experiments

C.1 Benchmark datasets and architectures

Datasets: In this work, we utilized various well-known image classification benchmarks. Below are the datasets, including their characteristics and original references.

Dataset	Dimensionality	Training set	Test set
MNIST	28×28 (grayscale)	60K	10K
CIFAR10	32×32 (RGB)	60K	10K
SVHN	32×32 (RGB)	600K	26K
ImageNet	224×224 (RGB)	1.3M	50K

The corresponding references for these datasets are LeCun et al. (1998); Krizhevsky & Hinton (2010); Netzer et al. (2011).

Neural network architectures: We evaluated various CNN neural network architectures, with details as follows:

Name	Layers	Loss function
LeNet-5	5	Cross-entropy
VGG11	11	Cross-entropy
VGG13	13	Cross-entropy
VGG16	16	Cross-entropy
VGG19	19	Cross-entropy
ResNet18	18	Cross-entropy
ResNet50	50	Cross-entropy
DenseNet121	125	Cross-entropy

The corresponding references for these architectures are Simonyan & Zisserman (2014); He et al. (2016); Huang et al. (2017); LeCun et al. (1998).

LeNet-5: The implementation for LeNet-5 was sourced from the following GitHub repository: `https://github.com/ChawDoe/LeNet5-MNIST-PyTorch/blob/master/model.py`.

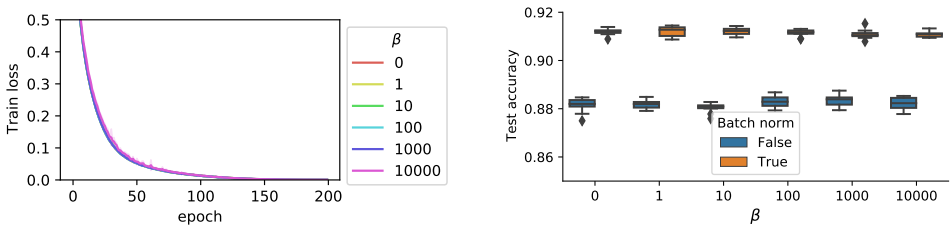
VGG: We used the PyTorch repository's implementation for the VGG models. It can be accessed at the following link: `https://github.com/PyTorch/vision/blob/main/torchvision/models/vgg.py`.

ResNet: For ResNet models, we utilized the PyTorch repository's implementation available at: `https://github.com/PyTorch/vision/blob/main/torchvision/models/resnet.py`. We made minor adjustments to the output layer's size (changing from 1000 to 10 classes) and the kernel size in the primary convolutional, varying from 7 to 3). When batch normalization was not used, we replaced the batch normalization layers with identity mappings.

DenseNet: The implementation for DenseNet was taken from the PyTorch repository, available at: `https://github.com/PyTorch/vision/blob/main/torchvision/models/densenet.py`.

C.2 Mitigating factor: Adam optimizer

After training a VGG11 network on CIFAR-10 using the Adam optimizer, we obtained results shown in Figure 11. Our findings are consistent with those presented in Section 3, but the network exhibits reduced sensitivity to β, resulting in improved stability of both test errors and training loss.



(a) 32-bit without batch normalization (b) 32-bit

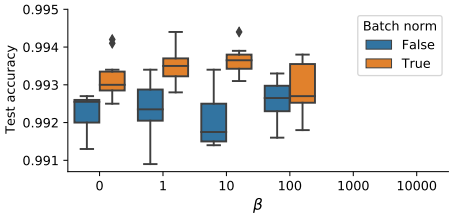
Figure 11: Training losses on CIFAR10 (left) and test accuracy (right) on VGG network trained with Adam optimizer and without batch normalization.

C.3 Additional experiments with MNIST and LeNet-5 networks

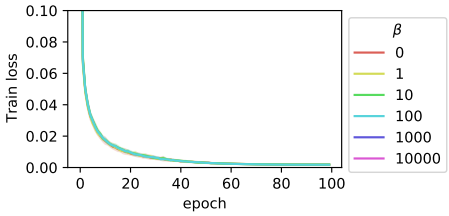
We repeated the experiments in Section 4.2 using a LeNet-5 network on the MNIST dataset. The results are depicted in Figure 12. We found that for 16 bits, the test accuracies were similar when training was possible, but $\beta = \{10^3, 10^4\}$ caused chaotic training behavior. For 32 bits, the test accuracies were mostly similar, except for $\beta = 10^4$. We noticed that the chaotic oscillations had completely disappeared.

C.4 Additional experiments with ResNet18

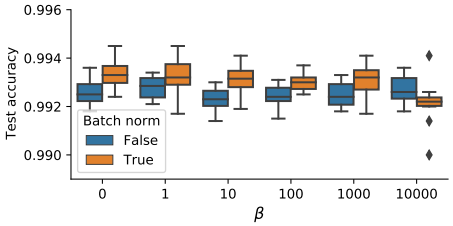
We performed the same experiments described in Section 4.2 using ResNet18 architecture trained on CIFAR 10. Figure 13 represents the test errors with or without batch normalization. For 16 bits, test accuracies are similar, but $\beta = 10^4$ induces chaotic training behavior. For 32 bits, test accuracies are identical, and the chaotic oscillations phenomena have entirely disappeared.



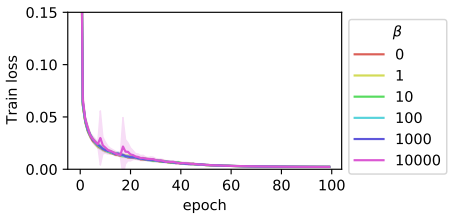
(a) 16-bit



(b) 16-bit, with batch normalization

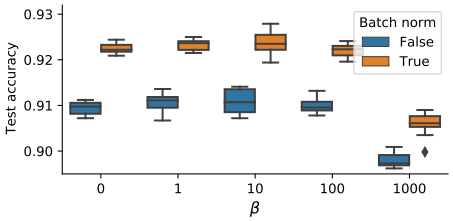


(c) 32-bit

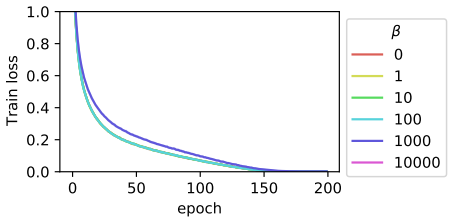


(d) 32-bit, without batch normalization

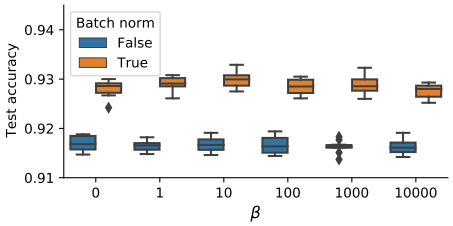
Figure 12: Training a LeNet-5 network on MNIST with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).



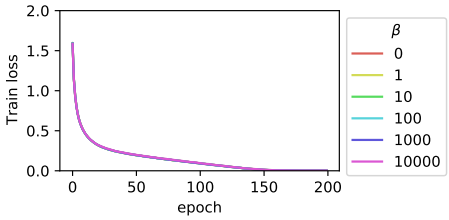
(a) 16-bit



(b) 16-bit, with batch normalization



(c) 32-bit



(d) 32-bit, with batch normalization

Figure 13: Training a ResNet18 network on CIFAR10 with SGD. We performed ten random initializations for each experiment, depicted by the boxplots and the filled contours (standard deviation).

C.5 Additional experiments with ResNet50 on ImageNet

We performed the same experiments described in Section 4.2 using a ResNet50 architecture trained on ImageNet. The test accuracy is represented in Figure 14. We employ mixed precision (Micikevicius et al., 2017; Jia et al., 2018), utilizing 16 and 32 bits precision to balance computational speed and

information retention. Test accuracies are similar when training is possible, but $\beta = 10^3$ induces chaotic training behavior.

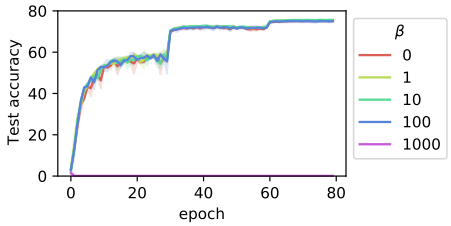


Figure 14: Test accuracy during training a Resnet50 on ImageNet with SGD using mixed precision. The shaded area represents three runs. We have a chaotic test accuracy behavior for $\beta = 10^3$.

D Complementary information

Computational Resources: All the experiments were conducted on four Nvidia V100 GPUs. This ensured consistent and reliable computation times across different experimental runs.

Code and Results Availability: The code corresponding to the experiments, as well as the results of these experiments, are publicly available. The repository can be accessed at the following URL: `https://github.com/AnonymousMaxPool/MaxPool-numerical`.

Licenses: The datasets used in our experiments are released under various licenses. CIFAR10 is under the MIT license, MNIST and SVHN are under the GNU General Public License, and ImageNet is under the BSD license. The libraries we used, Numpy and PyTorch, are released under the BSD license, while Python is released under the Python Software Foundation License.

Dataset	Network	Optimizer	Batch Size	Epochs	Time Per Epoch	Repetitions
MNIST	LeNet-5	SGD	128	100	2 seconds	10
CIFAR10	VGG11	SGD	128	200	9 seconds	10
CIFAR10	ResNet18	SGD	128	200	13 seconds	10
SVHN	VGG11	SGD	128	100	70 seconds	10
ImageNet	Resnet50	SGD	512	90	15 minutes	3

Table 5: Detailed experimental setup, including the dataset, neural network architecture, optimizer used, batch size, number of epochs, average computation time per epoch, and repetitions for each experiment.